The Code is the Model

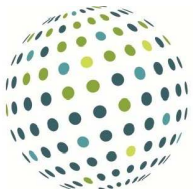
Meisser, Luzius

Abstract: Conventionally, agent-based models are specified in a combination of natural language and mathematical terms, and their implementation seen as an afterthought. I challenge this view and argue that it is the source code that represents the model best, with natural language and mathematical descriptions serving as documentation. This modeling paradigm is inspired by agile software development and adopting it leads to various - mostly beneficial - consequences. First, discrepancies between the specification documents and what the model actually does are eliminated by definition as the code becomes the specification. Second, replicability is greatly improved. Third, object-oriented programming is recognized as an integral part of a modeler?s skill set. Forth, tools and methods from software engineering can support the modeling process, making it more agile. Fifth, increased modularity allows to better manage complexity and enables the collaborative construction of large models. Sixth, the way models are published needs to be reconsidered, with source code ideally being part of the peer review. Seventh, the quality of source code in science is improved as it enjoys more importance, attention and scrutiny.

Originally published at:
Meisser, Luzius (2017). The Code is the Model. International Journal of Microsimulation, 10(3):184-201.

International Journal of Microsimulation (2017) 10(3) 184-201

International Microsimulation Association



The Code *is* the Model

Luzius Meisser

University of Zurich, Zurich, Switzerland
luzius@meissereconomics.com

ABSTRACT: Conventionally, agent-based models are specified in a combination of natural language and mathematical terms, and their implementation seen as an afterthought. I challenge this view and argue that it is the source code that represents the model best, with natural language and mathematical descriptions serving as documentation. This modeling paradigm is inspired by agile software development and adopting it leads to various - mostly beneficial - consequences. First, discrepancies between the specification documents and what the model actually does are eliminated by definition as the code becomes the specification. Second, replicability is greatly improved. Third, object-oriented programming is recognized as an integral part of a modeler's skill set. Forth, tools and methods from software engineering can support the modeling process, making it more agile. Fifth, increased modularity allows to better manage complexity and enables the collaborative construction of large models. Sixth, the way models are published needs to be reconsidered, with source code ideally being part of the peer review. Seventh, the quality of source code in science is improved as it enjoys more importance, attention and scrutiny.

KEYWORDS: AGENT-BASED MODELING, AGILE SOFTWARE ENGINEERING, MODELING METHODOLOGY

JEL classification: B41,C63,C88

1 INTRODUCTION

The specification and the implementation of computer programs were traditionally seen as distinct, sequential activities. Similarly, the specification and the implementation of agent-based models are usually seen as distinct, sequential activities. However, while software engineering moved on, agent-based modeling is still mostly done the traditional way. In modern software engineering, specification and implementation are closely integrated. Instead of completing the design upfront, the design is seen as emerging from incremental improvements of the program embodied by its code. There is no separate design document any more. Instead, the code is the design – an insight first described by Reeves (1992). I apply this insight to the process of agent-based modeling, leading me to question the conventional separation of specification and implementation and to boldly claim "the code is the model".[1]

Seeing the code as the model allows to build and refine agent-based models iteratively while programming, as opposed to writing down their specifications on paper first. It further implies that editing and adapting the model is done by editing and adapting its source code. The model as specified by the code becomes the primary deliverable, with accompanying papers focusing on the documentation of specific insights. This is in accordance with the principles of the *Agile Manifesto* by Beck et al. (2001), which had a profound impact on modern software engineering. Agile software development is a fundamentally pragmatist methodology that prioritizes people over processes, working software over documentation, collaboration over negotiation, and responding to change over plans.

Agile as a methodology is not undisputed. To a certain degree, it has become the victim of its own success. In a presentation titled "Agile is Dead", Thomas (2015), who is one of the authors of the agile manifesto, laments that some of the originally good ideas have been turned into rituals that are blindly followed without deeper understanding, thereby perverting the spirit of the manifesto. This paper sticks to the original insights of agile, with a focus on tight feedback loops and the cost of change. This focus is inspired by Fowler (2001), another author of the manifesto.

After diving deeper into the driving forces behind modern software engineering in section 2, each of the consequences listed in the abstract is described in its own section, concluding with the impact on software quality. They can be read in any order as the core insights are shortly repeated where necessary. The ideas and methods found in this paper are not intended to be followed blindly, but to serve as an inspiration on how to successfully build non-trivial agent-based models.

2 AGILE SOFTWARE DEVELOPMENT

"Probieren geht über Studieren" (Trying beats pondering).
- German proverb

While classic engineering focuses on ensuring high quality upfront, agile software development focuses on fast iterations and testing ideas quickly. At first sight, that looks like chaotic trial-and-error. It feels more comfortable to do traditional, sequential design with a clear separation between the specification and implementation stage. So what went wrong with software engineering? In order to understand agile development, one needs to consider the driving forces behind it, most notably the diminishing returns of upfront design.

The main driving force that makes it worthwhile to spend time on upfront design is the cost of correcting errors later (Fowler, 2001). An error in the design of a bridge can be extremely costly. Thus, it pays off to spend a lot of time upfront on planning before actually building the bridge. The question is: how does that change when the cost of building decreases? In the most extreme case of having robots that can automatically tear down and build bridges over night for free, one would certainly make use of that capability and rebuild the bridge a few times until it looks and functions as desired. Adding automatic testing of the bridge's properties brings this analogy quite close to the state of modern software engineering. When ideas can be tried out quickly with minimal effort, the returns of upfront planning diminish.

A second driving force behind modern software engineering is the dynamic environment in which it takes place. Evolving requirements and a continuous inflow of ideas further diminish the value of upfront planning. In the nineties, computer programs were supposed to last for years without updates and were built within the scope of software projects. A project is an organized endeavor with a start and an end. But web services like Google or Netflix are never finished, and thus do not fit the most basic premise of a project. Instead, they are continuously improved. Facebook's continuous delivery system pushes out an update every eight hours, as Rossi (2017) describes. This stands in stark contrast to the multi-year release cycle of traditional software. The ability to seamlessly deliver frequent and automatic updates decreases the cost of change further and allows to dynamically respond to evolving circumstances. In such an environment, "plans are nothing, planning is everything". This core insight of the agile manifesto was already formulated by Dwight Eisenhower, although in a different context.

A third important difference to classic engineering is that no one is constructing anything physical. Instead, both the architects and the programmers are engaging in a design activity, just at different levels of abstraction. This blurs the distinction between design and implementation, begging the question whether it makes sense for them to use different languages, different tools, and different processes. Since the language and tools for the implementation stage are given, the only option to get the high-level design and the low-level design closer is to increase the use of low-level tools in high-level design, for example by directly specifying a class hierarchy in source code instead of using UML diagrams. Attempts to do the opposite, namely to use high-level tools to do low-level design, tend to fail. The simple and effective high-level tools are not powerful enough to completely specify a software, and the high-level tools that are powerful enough tend to get more complex than the source code they were intended to replace. The cleanest way to formulate an algorithmic model is and stays plain source

code.

These three forces have contributed to the rise of *extreme programming* (Beck, 2000) and the creation of the agile manifesto (Beck et al., 2001). It acknowledges good design as something that emerges over time, and not something that can be fully specified upfront. The specification and the implementation phase are combined, thereby accelerating the development process and allowing for fast, agile iterations of collecting feedback and improving the software. It does not mean that there shouldn't be any design or other high-level planning, it just means that design and implementation should be closely integrated. Under agile software development, the source code is the only complete specification of the software. Other documents, such as protocol specifications, class diagrams or natural language descriptions, are still useful and often necessary to convey the high-level intent behind a program, but they are mere means to an end. The code is the primary deliverable and working software is the primary measure of progress.

3 MODEL SPECIFICATION

"An idea is nothing, its implementation everything."
- Alexander Kronrod (Landis, Yaglom, Brudno, Gautschi, & Senechal, 2002)

The insight that the source code of an agent-based model can serve as its specification is not new. Miller and Page (2007) state on page 76: "The actual computer code itself is a complete specification of the model, but there is a big difference between a complete specification and an accessible one." Furthermore, code is not academically publishable. This has lead researchers to seek for alternate ways of specifying agent-based models in their publications.

The most common approach is to specify the model in natural language, supported by mathematical equations where appropriate. One example of this approach is the financial leverage model by Thurner, Farmer, and Geanakoplos (2012). I mention this specific model because I reimplemented it and successfully reproduced its results. In that case, the exact sequence of events is not perfectly clear from the paper, exemplifying that it is very hard to write a complete specification without using programming languages. Still, the description is good enough to reproduce the results qualitatively with adequate effort. I successfully did the same for other simple agent-based models, indicating that this type of specification works reasonably well in simple cases. Accordingly, Miller and Page (2007) recommend to strive for simplicity in order to increase accessibility and to make sure the model can be fully specified within the scope of a paper. However, not all models of interest are simple.

Grimm et al. (2006) observe that the above approach often leads to unsatisfactory model descriptions, for example "not including enough detail of the model's schedule to allow the model to be re-implemented". As a remedy, they propose the ODD protocol and later refine it in Grimm et al. (2010). The ODD protocol specifies a program by creating two tables, one with all the variables and

one with all the functions. They believe that: "Once readers know the full set of (low-level) state variables, they have a clear idea of the model's structure and resolution." By doing so, Grimm et al. tear apart the object-oriented design, as they note themselves. The design of software often is hierarchical, with multiple encapsulated layers at different levels of abstraction, allowing the reader to focus on the functionality of interest. By pushing its components into a flat list, the essence of the program can get lost and its accessibility is reduced. Also, the ODD requirement of including every detail can make the specification overly lengthy. For example, Wolf et al. (2013) note that "a description of a Lagom model that strictly follows ODD would easily fill 50 to 80 pages." Like the other attempts before it, the ODD protocol suffers from the typical symptoms of trying to specify algorithmic models in something else than a programming language.

Accepting the futility of providing a complete specification of non-trivial models using the previously mentioned methods, about one in ten authors resort to pseudo-code (Janssen et al., 2017). Pseudo-code is well-suited to specify short algorithms of up to 25 lines of code within a paper. It is often used in computer science, for example by Flajolet, Fusy, Gandouet, and Meunier (2007), where the focus is on the mathematical analysis of algorithms as opposed to building software. However, pseudo-code is impractical for larger models and I advise against creating pseudo-code attachments. There is no guarantee that it matches the actually executed code. For example, Gualdi, Tarzia, Zamponi, and Bouchaud (2015) follow this practice, but the parameter $\delta = 0.2$ in the pseudo-code does not match the actual value from the source code, where it is 0.02.[2] Furthermore, it is questionable whether pseudo-code is more accessible than well-written code in a real programming language. In the end, the original code is the ultimate reference. That is also why many journals encourage providing source code as supplementary download, a practice followed by another 10% of authors according to Janssen et al. (2017). Ideally, the code resides in a browsable web repository, so it can be inspected without downloading or installing anything.

Agent-based models are fundamentally algorithmic and often of non-trivial size. Furthermore, they can be very sensitive to small changes, so providing incomplete specifications is not an option. Therefore, the cleanest way to specify agent-based models is to use source code. All other artifacts, such as flow charts, UML diagrams, natural language descriptions, ODD tables, and mathematical equations can be suitable means to describe the model at a higher level, but not to fully specify it. If an interested reader wants to know about the complete model in all its details, she should consult the source code, with the paper serving as a guide. "In software development, the design document is a source code listing" (Reeves, 2005).

4 REPLICABILITY

Replicability is the foundation of empirical science. It enables the independent verification of experiments and to reliably build new theories on top of existing knowledge. Despite its importance, the

replication of results often is harder than it should be. Chang and Phillip (2015) try to reproduce the results of 67 economics papers published in a selection of 13 reputable journals. They could only replicate 33% of them on their own, and an additional 10% with the authors' assistance. The primary reason for a failure to replicate results was missing software or data – even for journals that in theory have a policy of requiring source code and data. This section discusses how seeing the code as the model improves replicability and provides a replicability checklist.

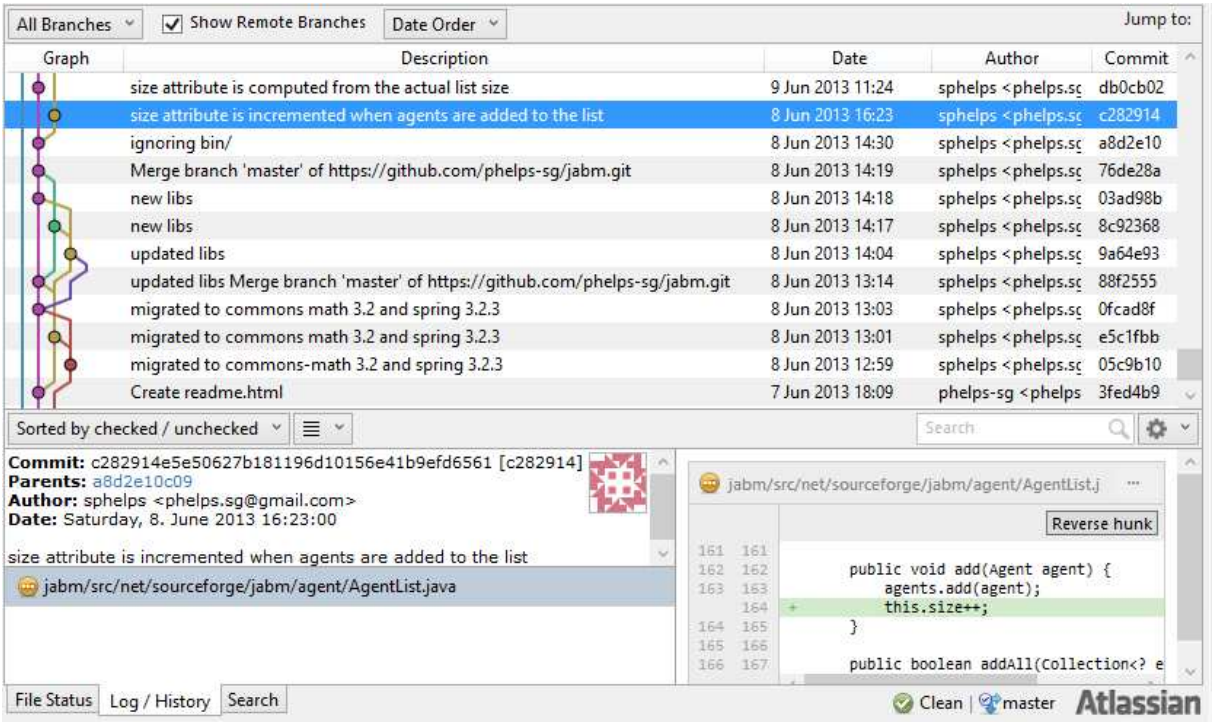
By seeing the code as the model, the code becomes the specification. Thereby, replicating an agent-based experiment becomes trivial in theory. All that needs to be done is compiling and running the code. The tedious and error-prone step of translating a natural language specification found in a paper into source code can be skipped.[3] However, the devil is in the detail. The author needs to properly prepare and document the software and its configurations in order for this to work smoothly. When running the simulation, the exact results reported in the paper should be deterministically reproduced. Axtell, Axelrod, Epstein, and Cohen (1996) call this level of replication *numerically identical*.

To go beyond numerically identical replication, one could even make sure that not only the raw numerical results are automatically reproduced, but also the charts and statistics shown in the paper. While this is a nice feature for readers and reviewers when playing with the simulation, its main benefit is for the author herself. In practice, she is often the one who is most likely to later reproduce the results. For example, when refining a paper or preparing a presentation, the ability to regenerate the charts of interest or slight deviations thereof with a few clicks can save a lot of time. Good replicability is not only desirable as a final outcome, but also accelerates the modeling process itself as it reduces the cost of trying out model variants and comparing them with each other.

The key tool for replicability is a good version control system. My system of choice is Git, which Bruno (2015) reviews in the context of computational economics. A version control system keeps a history of all changes made to the source code. Each time the programmer commits a change to the repository, a snapshot is made and the history grows by one entry. The space needed for a single snapshot usually is negligible, as only the difference to the previous snapshot is stored. Ideally, the programmer comments each commit, thereby implicitly creating a lab journal. Keeping a lab journal is also recommended by Miller and Page (2007). Figure 1 shows a screenshot of browsing the commit history of JABM written by Steve Phelps, one of the authors that publish their source code in a public Git repository. Today, commit comments are mostly technical, but they would also be an excellent way to shortly document scientific considerations.

By default, each step in the change history is identified with a unique fingerprint. To make them more accessible, significant milestones of a program are usually labeled with a human-readable tag. Tags can also be used to switch between different versions of a software or to compare them. When working with an agent-based model, each time the model is used to generate significant results, that particular version of the model including its inputs and outputs should be tagged. That way, the latest version of the model can be changed and improved without the risk of breaking the replicability of older results.

Figure 1: A Git client



This screenshot shows how the Git client SourceTree lists changes Steve Phelps made to the Java Agent-Based Modeling library (github.com/phelps-sg/jabm), which was used to produce the results presented in Caiani et al. (2014) and Caiani et al. (2015). The selected commit to the yellow branch with fingerprint c282914 apparently fixed the size counter of a list of agents named AgentList. When not only commenting on technical decisions, but also modeling considerations, Git can serve as electronic lab journal that is tightly linked to the relevant source code.

They are preserved as their own, tagged snapshots.[4]

Authors should adhere to the following checklist to ensure replicability and to increase accessibility:

1. Your paper must contain a high-level description of the model.
2. Your paper must link to the model's source code. Preferably, the code is hosted in a browsable web repository such as github.com.
3. Along with the link to the code, the fingerprints (hash) of the discussed versions should be provided. This proves that the code was not changed after the submission of the paper.
4. Your code must include a readme file with instructions on how to compile and run the simulation. This should include the program inputs and the expected outputs for each discussed result.
5. Your simulation should be deterministic. Running the same configuration twice should yield the exact same result.
6. You should specify under which conditions the code can be reused, for example under the MIT license (MIT, 1988). Academic use under the condition of proper attribution must be permitted.
7. You should encourage others to clone your model into their own repositories in order to im-

prove long-term availability. Prominently add the title of your paper to the readme file so the repository can be found with a web search even when the original links are broken.

8. The tools and libraries required to compile and run the software should be freely available for academic use. For example, Jupyter notebooks should be preferred over Mathematica notebooks.

9. You are encouraged to cross-reference the paper from the code and vice versa. In particular, you should make clear how the variable names from the paper (usually single-lettered) map to the ones in the code (should be long and descriptive).

10. Keep your model simple and accessible by following the hints given in Section 5.

5 OBJECT-ORIENTED PROGRAMMING

> "Programming frees us to adapt the tool to the problem rather than the problem to the tool."
> - Leigh Tesfatsion (2006)

While models can be specified in any sufficiently powerful language, its choice visibly impacts the end result, even when the model's motif stays the same. Generally, the cleanest way to specify equation-based models is to use mathematical terms, and the cleanest way to specify algorithmic models is to use a programming language. This section discusses how object-oriented design and good programming can make a model's specification more accessible.

The family of object-oriented languages seems a particularly good fit to formulate agent-based models. Dahl and Nygaard (1966) invented the first object-oriented programming language *Simula* to provide humans with an intuitive abstraction to write simulations. Incidentally, objects also are an excellent abstraction to manage complexity by encapsulating separate concerns, making object-orientation the most popular programming style by far today. As Tesfatsion (2006) points out, object-orientation resembles agent-based modeling. Individual agents act in accordance with private beliefs, which they update by observing local information. Both agents and objects are about encapsulating the state of individual entities, putting data and the functions that operate on that data together. Object-orientation fits very well with agent-based modeling and provides proven abstractions for managing complexity.

Besides supporting object-oriented design, today's programming languages also operate at a higher level of abstraction than older languages. Memory management and pointer arithmetics are not necessary any more, increasing accessibility. Also, object-orientation helps with the hierarchical organization of the software. Nonetheless, the programming skills of a modeler can make an enormous difference in accessibility. While it often takes programmers many years of experience to fully develop their skills, there are some basic rules that help writing simple and accessible code:

1. Keep your code clean and simple. Its purpose is not to show how smart you are, but to provide

an accessible formulation of your model to others and your future self. Given two options, choose the one that astonishes your readers the least.

2. Split code into small units with descriptive names. For example, one should split large functions into multiple small ones even if they are only called from one place. The purpose of a function is not to enable the reuse of its code, but to structure the program nicely. The same applies to classes.

3. Choose the simplest tool that does the job. You won't need gimmicks like dependency-injection frameworks or aspect-oriented programming. They are often poorly supported by the development environment and tend to make relevant information less accessible.

4. Avoid premature generalization: do not write a function to draw polygons if all you need for now is rectangles.

5. Avoid premature optimization: Your first priority is to get things right. You can still optimize later if necessary. Most of the time, it is not. "Premature optimization is the root of all evil" (Knuth, 1974).

6. "Favor object composition over class inheritance" (Gamma, Helm, Johnson, & Vlissides, 1994). Beginners tend to overuse inheritance in object-oriented programming. Prefer composition instead.

7. Choosing a popular programming language increases accessibility. As a social scientist, you do not need to be at the bleeding edge of computer science, you can relax and build on proven technology that is broadly understood.

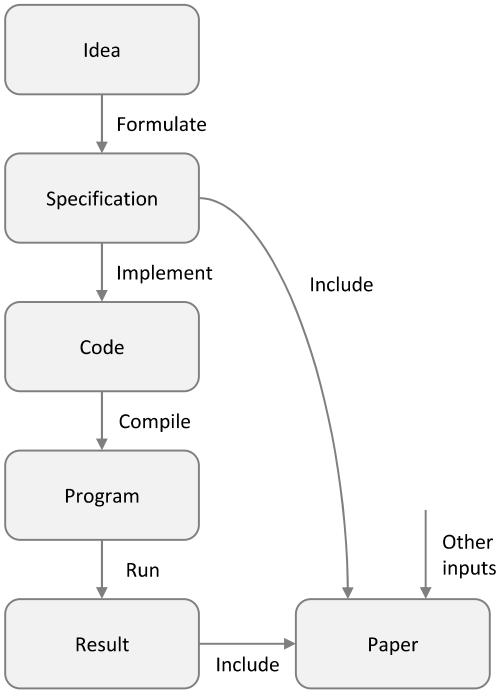
8. Avoid low-level languages such as C or Fortran. Java and C# are similarly fast and have fewer pitfalls. Python and other dynamically typed languages can be an excellent choice for small projects below 1000 lines of code, but are usually an order of magnitude slower — a personal observation that is confirmed by benchmarks such as Gouy (2017).

9. For larger projects, prefer a statically typed language, allowing you to painlessly refactor (restructure, rename, move, etc) your code as the model evolves (Fowler & Beck, 1999).

10. Avoid cargo cult programming: do not blindly follow conventions without understanding them. This includes everything you read in this paper.

Agent-based modelers with moderate experience should be able to formulate their models in code such that it is accessible to other researchers with a similar skill-level. Source code cannot replace high-level descriptions, but it should be the preferred way to provide the full model specification. Keeping code clean and concise requires some effort from the author, but pays off quickly thanks to lower costs of change and increased agility.

Figure 2: Waterfall modeling.



Notes: This diagram depicts the *waterfall* view of the modeling process of a computer simulation and its accompanying paper. It is highly idealized and hard to follow in practice. There is no indication on how to handle errors or how to otherwise refine the initial idea. Yet, this is often how the development process is implicitly assumed to work.

6 MODELING PROCESS

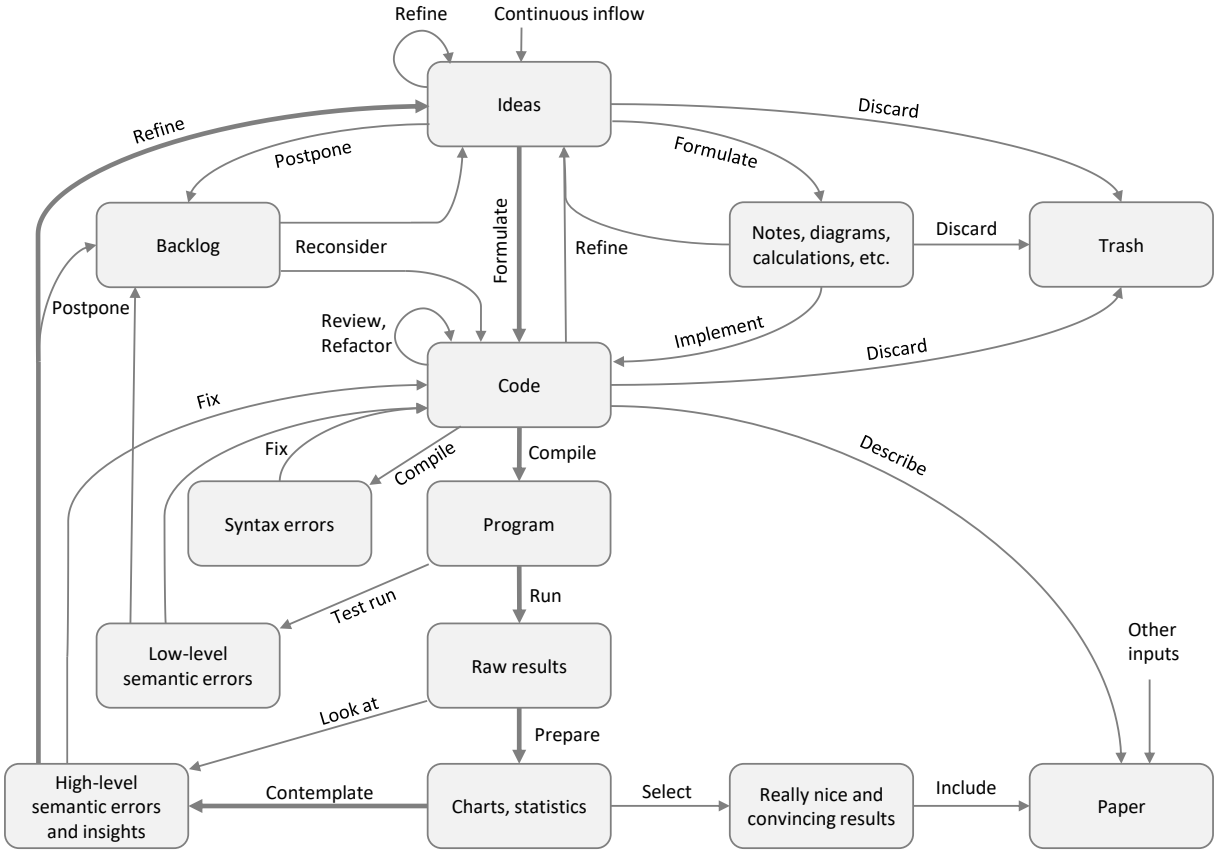
"The only way we validate a software design is by building it and testing it. There is no silver bullet, and no 'right way' to do design. Sometimes an hour, a day, or even a week spent thinking about a problem can make a big difference when the coding actually starts. Other times, 5 minutes of testing will reveal something you never would have thought about no matter how long you tried. We do the best we can under the circumstances, and then refine it."

- Jack Reeves (2005)

Seeing the code as the model allows to fully leverage all the tools and methods of agile software engineering in the modeling process, a selection of which is described in this section. The key is again to decrease the cost of change, thereby improving agility. This is mostly done by creating tight feedback loops to accelerate the detection of errors at all levels of abstraction, thereby enabling the modeler to identify and refine her best ideas faster.

The traditional modeling process is illustrated in Figure 2. The underlying assumption is that the design and the implementation are separate. Specification documents are created upfront and translated into source code at a later stage. The advantage of this separation is a higher degree of specialization. The designers of the model do not need to be fluent programmers. However, the price of this separation is a slower modeling and refinement process. Also, it comes with a higher risk of ideas getting lost in translation, as Wilensky and Rand (2007) already pointed out. The cycle from idea to specification,

Figure 3: Agile modeling.



Notes: This diagram presents a more realistic view of the modeling process, incorporating tight feedback loops to decrease the costs of testing ideas. Formal specifications got replaced with a set of temporary documents that help communicating and refining ideas. Unlike before, ideas can also be formulated directly in a programming language. From there, three feedback loops around different classes of errors help to decrease the time it takes to ensure the code works as intended. Once it does, the simulation outcome is used to refine the initial idea, closing the outer, bold feedback loop. Each time it is completed, the modeler learns something about her model and assumptions.

to implementation, to validation, and then back to the idea stage of the next version of the software takes a lot of time and resources. Accordingly, Miller and Page (2007) state on page 252: "An error in the design stage costs ten times more to correct in the coding stage and a hundred times more to fix after the program is in use." Consequently, traditional engineering tries to avoid having to return to the specification stage and aims at getting things right on the first attempt. In practice, this is rarely the case.

In contrast to that, agile software engineering embraces the aforementioned cycle and minimizes the cost of going through it. If done right, the above quote from Page and Miller no longer holds. Figure 3 depicts a more realistic modeling process that incorporates these insights. It allows to test and refine ideas much faster than before. Key is the creation of tight feedback loops around the various classes of errors and to automate them as much as possible. The modeler is still encouraged to refine ideas upfront, maybe with the help of diagrams, calculations, notes, charts, and other temporary documents that aid the thought process. But she is also free to try out ideas by adjusting the code directly. The feedback loop from idea to code, to validation and back to refining the idea is much tighter. However, this must not be understood as an invitation to do quick and dirty coding. Instead, code must be tidier

and better structured than under the traditional process. Otherwise, the gained agility is quickly lost again as the complexity of the codebase grows out of hand.

First, time must be invested into regularly refactoring the code, keeping the code clean, accessible, and modular (Fowler & Beck, 1999). Fortunately, IDEs such as Eclipse allow to seamlessly rename variables, move classes, extract interfaces, and perform many other refactorings without much manual labor. One should also not hesitate to remove unused parts of the model. If needed again later, they can be restored from the versioning system. Refactoring aims at improving the design of the code, thereby allowing to perform changes (including further refactoring) at a lower cost.

Second, one should automate the feedback loops around all three classes of errors shown in Figure 3 as much as possible. This is trivial for syntax errors that are detected on the fly and thus can be corrected within seconds. At the next level, the modeler should create a suite of unit tests to verify the behavior of individual classes. They should be triggered automatically whenever a change is made, such that the programmer is notified within seconds after introducing an error. However, it also takes some effort to write and maintain good units tests, so there is a trade-off.[5] At the very least, one should create a test that simply runs the simulation after every change. This works especially well when making extensive use of assertions as described next.

Third, one should follow the fail fast principle discussed in Shore (2004). Errors are much easier to detect in programs that fail hard and visibly upon encountering something unexpected than in programs that silently ignore the unexpected. For example, when implementing a function that operates on a list under the implicit assumption that this list is ordered, one should check the ordering in the beginning of the function and let the program crash in case of a violation. The alternative of letting the function ignore such a problem might lead to subtle and hard to detect errors. Another example would be a test for stock-flow consistency in an economic model after every simulation step. Assertions combine very well with automated tests, enabling the detection of errors within seconds after having unwittingly introduced them.

While automated testing allows the code to quickly converge towards its intended behavior, frequent and automated tests of scientific models can introduce the risk of overfitting. Non-trivial model are especially susceptible to this problem as they often have countless adjustable parameters. Sometimes, this can be addressed by testing the outcome against abstract theories, based on which one can create an unlimited number of test cases. For example, one could test whether prices in an economic simulation converge towards what classic theory predicts. Another good idea is to test the behavior of each agent on its own for certain reasonable criteria. That way, one can decrease the degrees of freedom significantly before testing the whole model against empirical data or the expectations of the researcher.

For larger software projects with many stakeholders, the process outlined in Figure 3 is too simplistic. I intentionally skipped practices like Scrum that are primarily concerned with the organization of teams

and the rhythm of iterations. For individual researchers, they are less relevant. Researchers are usually their own managers and customers, diminishing the need for formalizing their communication. I refer interested readers to *The Cathedral and the Bazaar* by Raymond (1999), which introduced the principle "release early, release often", Schwaber and Beedle (2002) for an introduction to Scrum, and Meyer (2014) for a more broad and critical review of agile methods.

7 MODULARITY

The hierarchical structure of source code is better suited for describing large, modular systems than the linear structure of a paper or the monolithic structure of equation systems. And good modularity is the key to manage complex systems. Well-modularized systems are more accessible, more robust and easier to change than monolithic systems. Like object-orientation, modularity is about encapsulation, but at a higher level of abstraction. At this higher level, the rules of good design can be different, and the object-oriented principle of keeping data and functions together is less important.

In economics, the difference between the structure of agent-based models and the structure of equation-based models resembles the difference between decentralized and centralized planning in *The Use of Knowledge in Society* by Hayek (1945). With equation-based approaches, a model consists of a monolithic equation system that is solved by a central planner. However, just like Hayek's central planner in the real world, such models struggle at incorporating diverse knowledge and are usually based on radically simplified assumptions. Farmer and Foley (2009) criticize conventional economic models as follows: "Even if rational expectations are a reasonable model of human behavior, the mathematical machinery is cumbersome and requires drastic simplifications to get tractable results." The design of large-scale systems that are easy to change is futile without good modularity.

In contrast to mathematical models, algorithmic models are much more versatile and better at embracing complexity. A clean separation of concerns allows to implement and test modules independently, without having to care much about the rest of the program, thereby significantly reducing the mental load of the programmer. In a well-modularized system, the consequences of a local change should be locally contained and not render the whole model unusable. Proper encapsulation ensures that a change at one end of the model does not adversely impact something seemingly unrelated at the other end. With monolithic mathematical models, often the opposite is the case.

Anthropologists would probably say that mathematics is a high-context culture, while software engineering is a low-context culture. A typical mathematical equation with its single-letter variables and functions requires a high level of context for correct interpretation, whereas software engineering encourages the use of long descriptive names and the creation of small digestible units of thought that can be processed on their own. Generally, low-context systems are more accessible to outsiders as everything is explicitly stated, thereby reducing the amount of prior knowledge required for understanding and adjusting them.

Besides helping both the human brain as well as the computer at handling complexity, modularization also enables the collaborative creation of large-scale models. With monolithic models, every contributor needs to understand everything. With modularized models, contributors just need to understand how to interact with the rest of the model and can otherwise treat it as a black box. Modularization also facilitates the reuse of components by others, which usually is impractical at the scale of objects. But similar to object-orientation, the primary goal of modularity is not to enable reuse, but to structure the software well.

8 PUBLISHING

What are the implications of seeing the code as the model for publishing? While journals can continue their established format and focus on the publication of insightful papers, they should make sure that the underlying source code is openly available. Good journals should also ensure that papers based on agent-based models follow the replicability checklist from section 4. Furthermore, replicability should be verified as part of the peer-review process, and ideally also the code reviewed. In the following, these recommendations are discussed in more detail.

As reasoned in section 3, the specification of an agent-based model is its code. As such, it must be provided to the readers. Chang and Phillip (2015) recommend making the provision of source code and data a strict condition for publication in all journals. However, code does not fit the format of journals and should thus be provided externally in a suitable web-based repository. Publishers do not have the infrastructure to properly host source code and some have a questionable track record when it comes to keeping links alive.[6] Thus, source code should be hosted with a purpose-built service like Github, which also allows to comfortably browse the code online. The easier it is to gain insights by inspecting source code, the more researchers will do so.

Unlike blogs and other native web-formats that are first published and ranked later by services such as Google, academic papers are first reviewed and then published. During the review process, experts decide in advance which papers are noteworthy and which are not. Academic journals provide a curated selection of papers that adhere to a high standard, with replicability being an important part of that standard. Consequently, reviewers should try to replicate the simulation results and journals should refuse the publication of papers whose results cannot be replicated with reasonable effort. Sometimes, a reviewer might not be able to run the simulation due to hardware or other practical constraints, but such cases should be the exception.

Services such as zenodo.org allow authors to create DOIs (digital object identifiers) for their source code and data. So in theory, it would be possible to refer to code directly. This can make sense when discussing an implementation detail that can only be found in the code. But generally, one should cite articles and not source code. This is also often preferred by the authors. For example, the authors of JAS-mine ask their users to cite Richardson and Richiardi (2016) when using their software, and not

the code itself.

Overall, seeing the code as the model is largely compatible with today's established publishing processes and in line with the trend of requiring the provision of source code when publishing an article based on an algorithmic model.

9 QUALITY

In agent-based modeling, high code quality is particularly important because agent-based models often react sensitively to small changes. As complex systems, they are susceptible to subtle errors. That makes it difficult to be confident whether an observed phenomenon represents a fundamental insight into the simulated matter or whether it is just an artifact of a low-level design decision. The best way to increase confidence in the model is to strive for high-quality code that yields replicable results and that is easy to change. The lower the costs of change, they easier it is to test the robustness of the results. If agent-based modeling wants to move beyond generating vague, stylized facts, there is no way around high-quality code.

As an economist, I am trained to consider the incentives. Today, there are barely any incentives to write well-structured code beyond what is necessary to get the job done. The traditional solution to this problem is to detach the specification from the code and to provide it in a separate, more visible document, so that the structure of the model is scrutinized as part of the traditional review process. As discussed in section 3, such specification are rarely complete and accurate in practice. The better solution is to direct more attention to source code, thereby creating a strong additional incentive for the author to structure it well and in an accessible way.

The measures discussed in the previous section 8 already are a good first step for drawing more attention to a model's code. When authors know that their chances of publication are increased by providing source code of high quality, they will try to do so. Ensuring replicability and accessibility starts to pay off beyond satisfying the intrinsic motivation of the author. Besides that, I expect models to increase in quality when their code is reviewed. Code reviews are widespread in software engineering and have been shown to decrease the defect rate significantly (Kemerer & Paulk, 2009).

Once researchers start to pay attention to each other's source code, a set of best practices and design patterns will emerge as they learn and copy from each other, further increasing accessibility and quality in general. Common design patterns help in reviewing code faster, just as reviewer today can skip well-known mathematical proofs that they have seen many times before. This creates a positive feedback loop of more attention leading to better code, and better code allowing the attention of reviewers to be used more effectively. Hopefully, seeing the code as the prime way to formulate a model contributes to the advent of a new generation of high-quality agent-based simulations that yield more useful and better verifiable results.

ACKNOWLEDGEMENTS

I would like to thank the participants of the 6th World Congress of Microsimulation as well as the participants of the Colloquium for Doctoral Students at the Institute for Banking & Finance of University of Zurich for their constructive critique and interesting discussions. Also, I would like to thank Paul Sevinç for reviewing this paper from a software engineering perspective.

REFERENCES

Axtell, R., Axelrod, R., Epstein, J. M., & Cohen, M. D. (1996). Aligning simulation models: A case study and results. *Computational & mathematical organization theory*, *1*(2), 123–141.

Beck, K. (2000). *Extreme programming explained: Embrace change*. Addison-Wesley.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., … others (2001). *The Agile manifesto.* Retrieved from `http://agilemanifesto.org`

Bruno, R. (2015). Version control systems to facilitate research collaboration in economics. *Computational Economics*, 1–7.

Caiani, A., Godin, A., Caverzasi, E., Gallegati, M., Kinsella, S., & Stiglitz, J. E. (2015). Agent based-stock flow consistent macroeconomics: Towards a benchmark model. *Available at SSRN*.

Caiani, A., Godin, A., Kinsella, S., Caverzas, E., Russo, A., Riccetti, L., … Gallegati, M. (2014). Innovation, demand, and finance in an agent based-stock flow consistent model. In *Social simulation conference*.

Chang, A. C., & Phillip, L. (2015). Is economics research replicable? Sixty published papers from thirteen journals say 'usually not'. *Finance and Economics Discussion Series*, *083*.

Dahl, O.-J., & Nygaard, K. (1966). Simula: An ALGOL-based simulation language. *Communications of the ACM*, *9*(9), 671–678.

Farmer, J. D., & Foley, D. (2009). The economy needs agent-based modelling. *Nature*, *460*(7256), 685–686.

Flajolet, P., Fusy, É., Gandouet, O., & Meunier, F. (2007). Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Analysis of algorithms 2007 (aofa07)* (pp. 127–146).

Fowler, M. (2001). The new methodology. *Wuhan University Journal of Natural Sciences*, *6*(1), 12–24.

Fowler, M., & Beck, K. (1999). *Refactoring: Improving the design of existing code*. Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

Gatti, D. D., Desiderio, S., Gaffeo, E., Cirillo, P., & Gallegati, M. (2011). *Macroeconomics from the bottom-up*. Springer Science & Business Media.

Gouy, I. (2017). *The computer language benchmarks game.* Retrieved from `https://benchmarksgame.alioth.debian.org`

Grimm, V., Berger, U., Bastiansen, F., Eliassen, S., Ginot, V., Giske, J., … others (2006). A standard protocol for describing individual-based and agent-based models. *Ecological modelling*, *198*(1), 115–126.

Grimm, V., Berger, U., DeAngelis, D. L., Polhill, J. G., Giske, J., & Railsback, S. F. (2010). The ODD protocol: A review and first update. *Ecological modelling*, *221*(23), 2760–2768.

Gualdi, S., Tarzia, M., Zamponi, F., & Bouchaud, J.-P. (2015). Tipping points in macroeconomic agent-based models. *Journal of Economic Dynamics and Control*, *50*, 29–61.

Hayek, F. A. (1945). The use of knowledge in society. *The American Economic Review*, *35*(4).

Janssen, M. A., et al. (2017). The practice of archiving model code of agent-based models. *Journal of Artificial Societies and Social Simulation*, *20*(1), 1–2.

Kemerer, C. F., & Paulk, M. C. (2009). The impact of design and code reviews on software quality: An empirical study based on psp data. *IEEE transactions on software engineering*, *35*(4), 534–550.

Knuth, D. E. (1974). Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, *6*(4), 261–301.

Landis, E. M., Yaglom, I. M., Brudno, V., Gautschi, W., & Senechal, M. (2002). Remembering A.S. Kronrod. *The Mathematical Intelligencer*, *24*(1), 22–30.

Meyer, B. (2014). *Agile! The good, the hype and the ugly*. Springer.

Miller, J. H., & Page, S. E. (2007). *Complex adaptive systems: An introduction to computational models of social life*. Princeton university press.

MIT. (1988). *MIT license.* Retrieved from `https://choosealicense.com/licenses/mit/` ([Online; accessed 12-May-2017])

Müller, B., Balbi, S., Buchmann, C. M., De Sousa, L., Dressler, G., Groeneveld, J., … others (2014). Standardised and transparent model descriptions for agent-based models: Current status and prospects. *Environmental Modelling & Software*, *55*, 156–163.

Raymond, E. (1999). The cathedral and the bazaar. *Knowledge, Technology & Policy*, *12*(3), 23–49.

Reeves, J. W. (1992). What is software design. *C++ Journal*, *2*(2), 14–12.

Reeves, J. W. (2005). Code as design: Three essays. *Developer. The Independent Magazine for Software Professionals*, 1–24.

Richardson, R., & Richiardi, M. (2016). JAS-mine: A new platform for microsimulation and agent-based modelling. *International Journal of Microsimulation*, 106-134.

Rossi, C. (2017). *Rapid release at massive scale.* Retrieved from `https://code.facebook.com/posts/270314900139291/rapid-release-at-massive-scale/`

Schwaber, K., & Beedle, M. (2002). *Agile software development with Scrum*. Prentice Hall Upper Saddle River.

Shore, J. (2004). Fail fast. *Software, IEEE*, *21*(5), 21–25.

Tesfatsion, L. (2006). Agent-based computational economics: A constructive approach to economic theory. *Handbook of computational economics*, *2*, 831–880.

Thomas, D. (2015). *Agile is dead.* Retrieved from `https://pragdave.me/speak#agile-is`

-dead

Thurner, S., Farmer, J. D., & Geanakoplos, J. (2012). Leverage causes fat tails and clustered volatility. *Quantitative Finance*, *12*(5), 695–707.

Wilensky, U., & Rand, W. (2007). Making models match: Replicating an agent-based model. *Journal of Artificial Societies and Social Simulation*, *10*(4), 2.

Wolf, S., Bouchaud, J.-P., Cecconi, F., Cincotti, S., Dawid, H., Gintis, H., … others (2013). Describing economic agent-based models: Dahlem ABM documentation guidelines. *Complexity Economics*, *2*(1).

NOTES

[1]A more agreeable but less catchy variant of this claim is: "Source code is the best way to formulate an algorithmic model, just like equations are the best way to formulate a mathematical model."

[2]I thank Stanislao Gualdi for providing me with the source code, which allowed me to discover this discrepancy. The correct value $\delta = 0.02$ is also mentioned elsewhere in the paper, so this is only a typo in the pseudo-code listing.

[3]In contrast, Müller et al. (2014) write: "although the provision of source code technically facilitates model replication, it may circumvent the consistency check between the conceptual model and its implementation (one purpose of model replication) by encouraging replicators to simply copy the source code." Detecting errors in an experiment's description and in the interpretation of its outcome certainly is important, but independent of replication. Replication aims at doing what the original researchers actually did, and not what they intended to do.

[4]For larger simulations, it might also be worthwhile to not only work with tags, but to create entire branches for each publication. A branch forks the change history of a project into two paths and allows to adjust one branch of the project without affecting the other. That way, one can create a clean copy of the simulation that only contains what is strictly needed to reproduce the discussed results, making the code leaner and increasing its accessibility. Generally, branches allow a software to develop into two different directions or having versions that evolve at different speeds, making the development process more flexible. However, it is also a somewhat advanced feature and merging branches later can be troublesome. Tags already suffices in many cases.

[5]Some developers take testing to the extreme and do test-driven development, which requires to specify the behavior of each piece of software upfront through a test before writing its actual code. However, this can also go against the spirit of agile. Tests increase agility by reducing the risk of change, but they also reduce agility as changes need to be implemented twice. These developers often also demand "100% test coverage", which means that every line of code is executed at least once during testing. But that still does not guarantee that the code is correct, so it is a somewhat misleading metric. Also, one should note that in a majority of cases where a unit test fails, it is the test that needs to be adjusted, and not the actual code. This is a consequence of test code having lower quality requirements than model code.

[6]For example, Gatti, Desiderio, Gaffeo, Cirillo, and Gallegati (2011) encourage readers to visit www.springer.com/series/9601 for further information on the last page of their book, but Springer seems to have broken that link.